Additional Feature

Our additional feature allows the user to calculate an estimation of the infected livable area. The program prompts the user to enter a date and a zip code. Using the date and zip code, the program obtains the number of positive cases, which is then divided by the population of the zipcode. Using the zip code, the total livable area of the zipcode is calculated. Next the program multiplies the total livable area by the percentage of the population that has been infected, resulting in the estimated infected living area. Since multiple people who test positive could live in the same property or frequently travel between different properties, this method provides a rough estimate of the squared units that are infected as of a particular date in a specific zip code.

For example, the user enters the date 01-19-2021 and the zip code 12345. Let's say that the population of 12345 is 100 people, there are 8 positive cases in 12345, and the total livable area of all properties in 12345 is 5000 square units. The infected population is 8%. So our estimation of the infected living area is 8% * 5000 = 400 infected square units.

We verified that our additional feature was functioning properly by writing test files that should produce known results. When we keyed in a particular set of variables, we knew the expected results ahead of time. If the actual results did not match the expected results, then we knew there was an issue with our code. Upon receiving the expected results multiple times across multiple test files, we are confident that our feature is functioning properly.

Use of Data Structures

When reading population data, we store the PopulationData objects in a set to prevent duplicate data. Since each entry in the population.csv file has a unique zip code, using set would ensure each zip code does not have a duplicate total population. Using a set for population also allows a faster search in calculating data using population, since the time complexity for most set operations, including look up, is O(1).

Due to the complexity of the property files and the program's functions that used the property data, we decided to nest 2 data structures to pass the information from the PropertyReader to the PropertyProcessor. The PropertyProcessor calls the readPropertyFile() function, which returns a map with a String as the key and a List as the value.

As the property data is used to calculate the average market value and average livable area, a list seemed to be the logical choice as it allowed for iteration. While we briefly considered using a set, we opted for the List as a set does not allow for duplication. Duplication was a concern because the property file contained numerous fields that were not stored. We stored the zip code, market value, and total livable area. Theoretically, there exists a file in which 2 different properties could both have the same values for those fields, but a set would think that they are the same property (object). The exclusion of one of those properties would lead to incorrect results. Additionally, the time complexity for most List operations are O(1) or at worst O(n), which is still relatively quick. A List also allows for the preservation of order, which may be helpful for future project improvements.

A map with the zipcode (String) as the key would increase the efficiency of functions 3.4-3.6 because each of those functions is exclusively concerned with 1 user entered zip code. While a List of properties could have been returned from the reader with the same Big-Oh complexity of O(n), the map greatly reduces the sheer number of properties that would have to be examined by the processor when evaluating the averages.

For example, properties.csv has over 500,000 properties listed. If we had chosen to return a List from readPropertyFile(), then every time the user calculates the average market value or livable area all 500,000 properties would have to be checked to see if they were in the specific zip code. Thanks to the map's key being the zip code, now only the properties within the user specified zip code need to be examined to calculate the averages.

By nesting the List of properties in a map with the zip code as the key, the overall efficiency of the processing functions is increased and a lot of repetitive overhead work is avoided.

Lessons Learned

Once the instruction packet was made available to us, we each read through the packet in its entirety and scheduled a Zoom meeting to agree upon an approach to the project. Together, we discussed the overall design and how we would implement the N-tier architecture. We decided that the best course of action would be to have each person create some code in each tier. We did not feel it would be fair or beneficial to have one person do all the data management code or all the processing code. Since the Property Data file was more complex, one team member was responsible for the reading and processing of the property data. The other partner was responsible for the reading processing of population and covid data. We both collaborated on the code for main as well. Due to the relative simplicity of the UI tier and the Logging tier, one team member was responsible for the UI and the other handled the logger.

At the beginning of our partnership, we established clear methods of communication for us to abide by during the project. We decided to limit our methods of communication to prevent us from missing any messages or updates. We agreed that for all brief communications we would use Slack and that for detailed written communication we would use email. At least once a week we would gather in a Zoom meeting and/or Slack huddle to verbally discuss our progress, the challenges we experienced, and the changes we needed to make to the project.

For version control we decided to exclusively use GitHub to ensure that we each had access to the most updated version of the code. When either one of us updated the code on GitHub, we would make the other team member aware of the update via a slack message. It was up to each individual team member to ensure that they downloaded the most recent version of the code to their local repository. Using a version control system allows us to keep track of commits we have made, and roll back changes as necessary. In addition, git made it easy for us to collaborate on the program without stepping on each other's toes. When making changes simultaneously, we used git to review each other's changes and resolve conflicts when merging.

Overall, we think that our collaboration was successful. The environment we created for ourselves allowed each of us to safely express our concerns or difficulties without fear of judgment or undue criticism. This experience allowed us to examine another person's code, and by extension their thought and perspective, which allowed for a lot of learning. For example, we each implemented a different system to read the files. While both methods successfully read the files, seeing each other's approach allowed us to see another way to solve this problem.

Although we had roughly sketched out a timeline for the project, which detailed how long we thought each tier would take to complete, we did stray from that original timeline. As we started working on the project, we discovered that our original timeline overestimated our capabilities. Specifically, we thought that the code and testing for the data management tier would take less time; however, due to its complexity, it took more time and effort. We were both flexible enough to adjust our timelines and mature enough to understand that no one was at fault. Luckily, we had both thought ahead and the original timeline had a 5 day buffer, which we used to ensure the project's completion.

Throughout this challenging project, we feel that we rose to the occasion, met the expectations, and have learned important lessons about communication and collaboration that we will take with us as we grow as people, students, and in our careers.